Java: Multithreading

IF2210 – Semester II 2020/2021

Multithreading

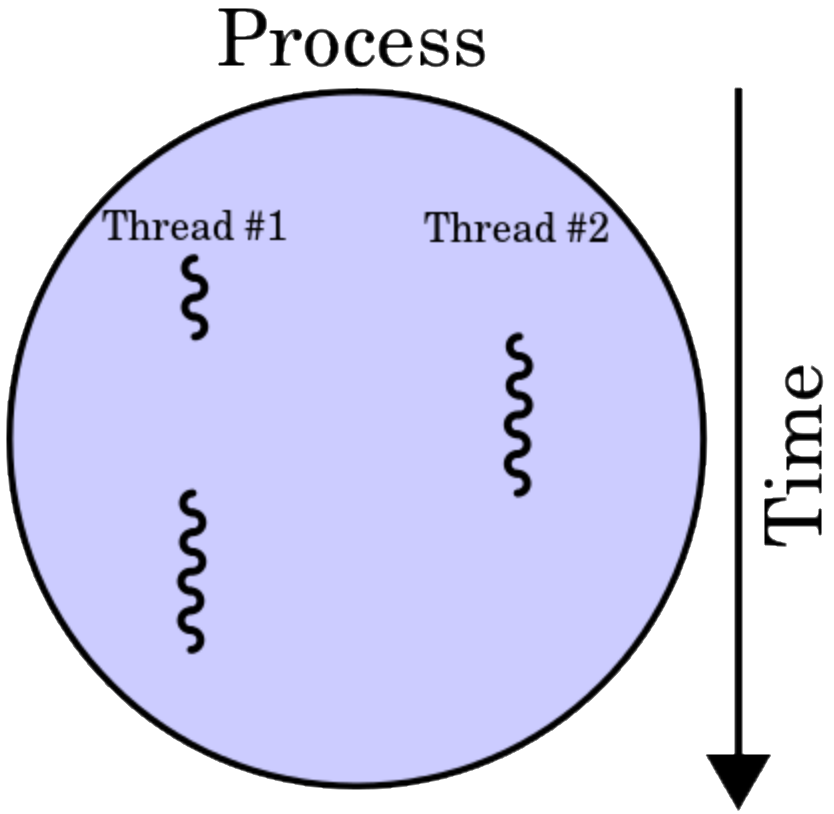
- Definition
  - *Multithreading is a type of execution model that allows **multiple threads to exist** within **the context of a process** such that they execute **independently** but **share their process resources.***

    https://www.techopedia.com/definition/24297/multithreading-computer-architecture

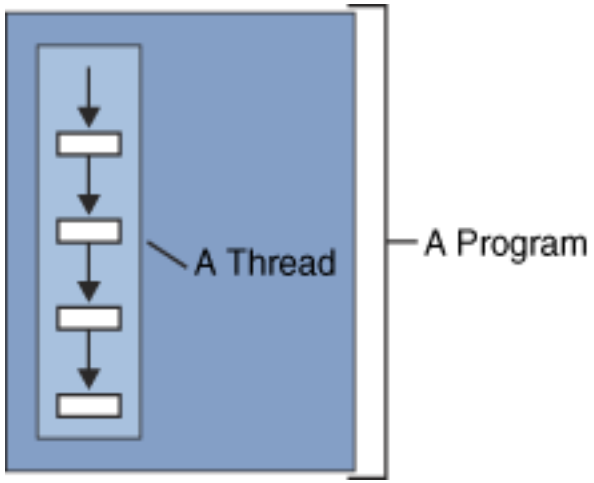
- What is a thread?
  - *A **thread** of execution is the **smallest sequence of programmed instructions** that can be managed independently by a scheduler.*

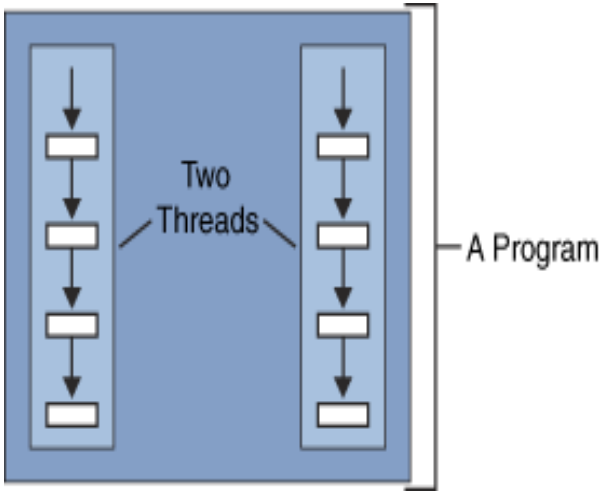
    https://en.wikipedia.org/wiki/Thread_(computing)

Process & thread



Single- vs. multithreading





Contoh

- *Web browser*
  - Mendownload halaman *web*
  - Menggambar *progress*
  - Menampilkan halaman
  - Menangani input *user*
- *Messenger*
  - Menerima input *user*
  - Menerima data dari *user* yang diajak *chat*
  - Menampilkan *user*

Kapan memakai Thread?

- Pada aplikasi GUI
  - Agar aplikasi tidak tampak "*hang*" ketika melakukan pemrosesan lama (data besar, atau proses memang lama)
- Aplikasi yang menangani banyak *event* dalam satu waktu
  - Server TCP/IP (Server Web, FTP)
- Aplikasi asinkron
  - *Chat*

Mengimplementasikan Thread

- Setiap *thread* berisi kode apa yang ingin kita jalankan.
- Ada dua cara membuat *thread*:
  - Menurunkan dari kelas `Thread`
    - *Override* method `run()`, defaultnya tidak melakukan apa-apa
  - Mengimplementasikan *interface* `Runnable`
    - Mengimplementasikan method `run()`
- Kapan men-*subclass* `Thread` dan kapan mengimplementasikan `Runnable`?

Thread vs Runnable

- Kelas hanya bisa diturunkan dari satu kelas lain.
- Kelas bisa mengimplementasikan banyak *interface*
  - Jika sudah meng-*extend class* lain, pasti memakai *interface*.
- `Thread` memiliki method yang berhubungan dengan *Thread*
  - Untuk setiap *interface* `Runnable` tetap butuh objek `Thread`.

Thread sederhana

```
class MyThread extends Thread {
    String s;

    public MyThread(String str) {
        s = str;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(s);
            try { Thread.sleep(1000); }
            catch (Exception e) {}
        }
    }
}
```

Menjalankan Thread

```
public class MultithreadingDemo1 {
    public static void main(String[] args) {

        new MyThread("Jamaica").start();
        new MyThread("Fiji").start();
    }
}
```

- Output: akan muncul "Jamaica" dan "Fiji" bergantian.

`Thread.sleep(int)`

- Menunda eksekusi selama *n* milidetik.
- *Delay* dalam kasus ini agar output lebih terlihat.
  - Agar lebih jelas, *delay* seharusnya random.
  - *Delay* yang *fix* akan membuat tampilan *Thread* berselang-seling.

Memakai Interface Runnable

```
class MyRunnable implements Runnable {
    String s;

    public MyRunnable(String str) {
        s = str;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(s);
            try { Thread.sleep(1000); }
            catch (Exception e) {}
        }
    }
}
```

Driver untuk Runnable

```
public class MultithreadingDemo2 {
    public static void main(String[] args) {

        Runnable r1, r2;
        r1 = new MyRunnable("Jamaica");
        r2 = new MyRunnable("Fiji");

        new Thread(r1).start();
        new Thread(r2).start();
    }
}
```

- Perhatikan bahwa objek `Thread` perlu diciptakan, dengan objek `Runnable` sebagai argumen konstruktornya.
- Output: akan muncul "Jamaica" dan "Fiji" bergantian.

Thread dengan Anonymous Class

```
public class MultithreadingDemo3 {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                while (true) {
                    System.out.println("Jamaica");
                    try { Thread.sleep(1000); }
                    catch (Exception e) {}
                }
            }
        };
        new Thread(r).start();
    }
}
```

Thread dengan Lambda Expression (Java 8)

```
public class MultithreadingDemo4 {
    public static void main(String[] args) {
        Runnable r = () -> {
            while (true) {
                System.out.println("Jamaica");
                try { Thread.sleep(1000); }
                catch (Exception e) {}
            }
        };
        new Thread(r).start();
    }
}
```

- Bacaan tentang lambda expression di Java:
  - http://tutorials.jenkov.com/java/lambda-expressions.html
  - https://medium.freecodecamp.org/learn-these-4-things-and-working-with-lambda-expressions-b0ab36e0fffc

Timer

- Selain `Thread`, Java menyediakan `Timer`.
- `Timer` bisa digunakan untuk menjadwalkan pekerjaan.
  - Tidak perlu membuat *thread* terpisah secara manual.
  - Menyederhanakan kasus tertentu.
- Apa yang bisa dilakukan oleh `Timer` bisa dilakukan oleh `Thread`.

Menjadwalkan pekerjaan

- Suatu pekerjaan bisa dijadwalkan agar berjalan di suatu waktu di masa depan.
  - Ada “sesuatu” (`Timer`) di latar belakang yang menghitung sampai waktu yang ditentukan.
- Menggunakan kelas `TimerTask` dan `Timer` di `java.util`

```
import java.util.TimerTask;
import java.util.Timer;
```

- Membuat kelas turunan `TimerTask`.

Memakai Timer & TimerTask

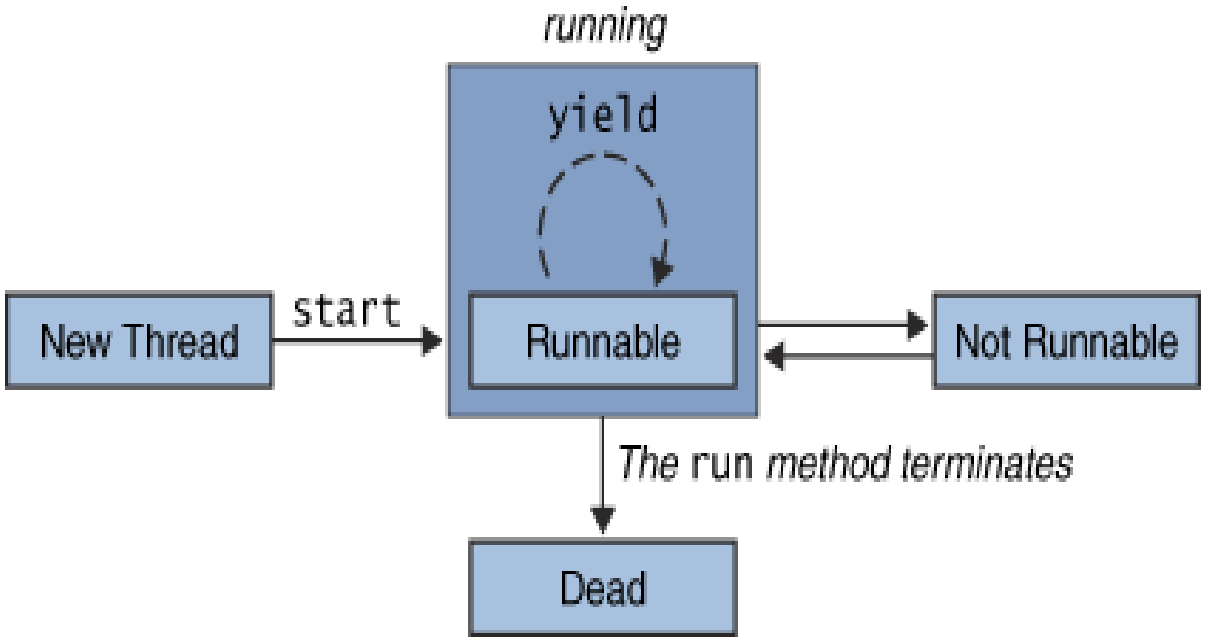
```
import java.util.Timer;
import java.util.TimerTask;

class MyTask extends TimerTask { // berisi pekerjaan yg dilakukan
    private String s;
    public MyTask(String str) { s = str; }
    @Override
    public void run() {
        System.out.println(s);
    }
}

public class TimerDemo {
    public static void main(String[] args) {
        Timer timer = new Timer();
        long delay = 1000, period = 2000; // in milliseconds
        timer.schedule(new MyTask("Once"), delay);
        timer.scheduleAtFixedRate(new MyTask("Repeated"), delay, period);
    }
}
```

Siklus Hidup Thread

- *Thread* hidup selama method `run()` masih berjalan



Thread bisa berhenti sementara

- `Thread` bisa di-*suspend* karena 3 hal:
  - Menunggu I/O:
    - Membaca dari jaringan atau dari *disk*.
  - Method `sleep` dipanggil untuk `Thread` tersebut
    - `Thread` akan berhenti selama waktu *sleep*.
  - Menunggu *event* karena pemanggilan `wait`
    - Akan dijelaskan pada sinkronisasi `Thread`.

Thread Priority/Scheduling

- `Thread` bisa diberi prioritas berbeda (min:1, max:10):
  - dari konstruktor
  - dengan method `setPriority()`
- `Thread` yang lebih tinggi prioritasnya lebih sering berjalan
  - Umumnya berarti akan lebih cepat selesai.

Masalah sinkronisasi

- Dua hal yang berjalan bersamaan akan sangat mungkin bertabrakan.
  - Misalnya:
    - *Thread* A akan menulis "Hello" lalu "World" ke file X.
    - *Thread* B akan menulis "Bye" lalu "Cruel World" ke file X.
    - Bisa jadi yang tertulis adalah "Hello" "Bye" "World" "Cruel World".
  - Contoh lain:
    - Akses variabel bersamaan.

Me-*lock* Object (1)

- Agar satu *method* selesai melakukan pekerjaannya sebelum diinterupsi tambahkan *keyword* `synchronized`:

```
public synchronized void write() { }
```

- Ketika suatu *method* synchronized di suatu objek dipanggil, objek itu akan "dikunci" agar tidak ada yang bisa mengakses *method* dalam objek tersebut yang sifatnya *synchronized*.

- Jika method tidak di-*synchronize*, gunakan:

```
synchronized (object) {
    // Statement yang akan dilakukan
    // dalam keadaan objek ter-lock
}
```

Me-*lock* Object (2)

- Perhatikan bahwa keduanya ini sama:

```
class Whatever {
    synchronized ReturnType method(...) {
        /*BODY*/
    }
}
```

```
class Whatever {
    ReturnType method(...) {
        synchronized (this) {
            /*BODY*/
        }
    }
}
```

Wait dan Notify

- Dalam permasalahan tertentu, terkadang *method* harus menunggu agar method lain selesai bekerja.
- Kasus klasik adalah: *Producer-Consumer* (paling serderhana hanya 1 *producer* dan 1 *consumer*).
- Ada sebuah *Producer*:
  - Menghasilkan Sesuatu.
- Ada sebuah *Consumer*:
  - Menghabiskan apa yang diproduksi *Producer*.

Contoh kasus

- Producer Pizza.
- Consumer Pizza.
- Objek PizzaBox (untuk meletakkan Pizza).
  - PizzaBox hanya bisa diisi satu pizza.
- Jika Pizza belum diambil, Producer tidak boleh menaruh Pizza lagi.
- Jika Pizza tidak ada, consumer harus menunggu Pizza.

Contoh implementasi

```
class PizzaBox {
    Pizza current;

    // mengambil pizza
    synchronized Pizza get() {
        return current;
    }

    // menaruh pizza
    synchronized void put(Pizza p) {
        current = p;
    }
}
```

```
class PizzaProducer extends Thread {
    PizzaBox pizzaBox;

    PizzaProducer(PizzaBox pb) {
        pizzaBox = pb;
    }

    @Override
    public void run() {
        while (true) {
            Pizza p = new Pizza(); // buat pizza

            // pizza sudah jadi, taruh di box
            pizzaBox.put(p);
        }
    }
}
```

```
class PizzaConsumer extends Thread {
    PizzaBox pizzaBox;

    PizzaConsumer(PizzaBox pb) {
        pizzaBox = pb;
    }

    @Override
    public void run() {
        while (true) {
            Pizza p = pizzaBox.get(); // ambil pizza

            // makan pizza ...
        }
    }
}
```

Main program

```
public class PizzaDemo {
    public static void main(String[] args) {
        PizzaBox box = new PizzaBox();
        new PizzaProducer(box).start();
        new PizzaConsumer(box).start();
    }
}
```

Masalah

- Implementasi tadi belum menangani kemungkinan-kemungkinan:
  - Ada Pizza, tapi Producer mencoba menaruh Pizza lagi: Pizza yang lama akan tertimpa.
  - Tidak ada Pizza, tapi consumer berusaha memakan Pizza: consumer mendapatkan Pizza yang lama.
- Harus ada variabel yang menandakan saat ini sedang ada Pizza atau tidak.

Revisi 1

```
class PizzaBox {
    Pizza current;
    boolean available = false;

    synchronized Pizza get() {
        while (!available) ; // Tunggu sampai ada
        available = false;   // sudah diambil
        return current;
    }

    synchronized void put(Pizza p) {
        while (available) ; // tunggu sampai diambil
        available = true;   // pizza siap diambil
        current = p;
    }
}
```

Masalah

- Versi ini menggunakan *busy waiting* (menunggu dengan cara *loop*).
- Ini harus dihindari karena membuang waktu CPU (tidak efisien).

Solusi: wait dan notify

- Ketika menunggu `Pizza`, `get()` harus menunggu sampai diberitahu bahwa `Pizza` sudah datang:
  - Dengan method `wait()`
- Ketika menaruh `Pizza`, `put()` harus diberitahu bahwa `Pizza` sudah diambil:
  - Dengan method `notify()`

Revisi 2

```
class PizzaBox {
    Pizza current;
    boolean available = false;

    synchronized Pizza get() {
        while (!available) {
            try { wait(); } // tunggu producer
            catch (InterruptedException e) {}
        }
        available = false; // sudah diambil
        notify(); // beritahu bahwa sudah diambil
        return current;
    }

    synchronized void put(Pizza p) {
        while (available) {
            try { wait(); } // tunggu sampai diambil
            catch (InterruptedException e) {}
        }
        current = p;
        available = true; // pizza siap diambil
        notify(); // beritahu pizza sudah siap
    }
}
```

wait, notify dan notifyAll

- `wait`
  - Menunggu sampai ada `notify()` atau `notifyAll()`
  - Mungkin menghasilkan exception `InterruptedException`
- `notify`
  - Memberi sinyal agar `wait` berhenti
  - Hanya memberitahu satu objek (yang mana? Tergantung implementasi Java)
- `notifyAll` (ini yang umumnya selalu dipakai)
  - Memberitahu semua objek yang menunggu

Tambahan

- Konkurensi merupakan masalah kompleks
  - *Deadlock*
  - *Starvation*
- Pelajaran Lengkap mengenai konkurensi diajarkan pada kuliah *Operating System*